5.3 The Python Memory Model: Introduction

In [1.4 Storing Data in Variables], we introduced the *value-based memory model* to help keep track of variables and their values:

Variable	*Value*
`distance1`	`1.118033988749895`
`distance2`	`216.14809737770074`

From this table we can surmise that there are two variables (`distance1` and `distance2`), each associated with a `float` value. However, now that we know about reassignment and mutation, a more complex memory model is needed: the *object-based memory model*, which we'll simply call the *Python memory model*, as this is the "standard" representation Python stores data.

Representing objects

Recall that every piece of data is stored in a Python program in an **object**. But how are the objects themselves stored? Every computer program (whether written in Python or some other language) stores data in computer memory, which you can think of as a very long list of storage locations. Each storage location is labelled with a unique memory address. In Python, every object we use is stored in computer memory at a particular location, and it is the responsibility of the Python interpreter to keep track of which objects are stored at which memory locations.

As programmers, we cannot control which memory addresses are used to store objects, but we can access a representation of this memory address using the built-in `id` function:

```
>>> id(3)
1635361280
>>> id('words')
4297547872
```

Formally, we define the **id** of a Python object as a unique `int` identifier to refer to this object.[1] Every object in Python has three important properties—*id*, *value*, and *type*—but of these three, only its *id* is guaranteed to be unique.

[1] The details of how Python translates memory addresses into the integers are not important to us.

In Python, a variable is not an object and so does not actually store data; variables store an id that *refers* to an object that stores data. We also say that variables *contain* the id of an object. This is the case whether the data is something very simple like an `int` or more complex like a `str`. To make this distinction between variable and objects clear, we separate them in different parts of the Python memory model.

As an example, consider this code:

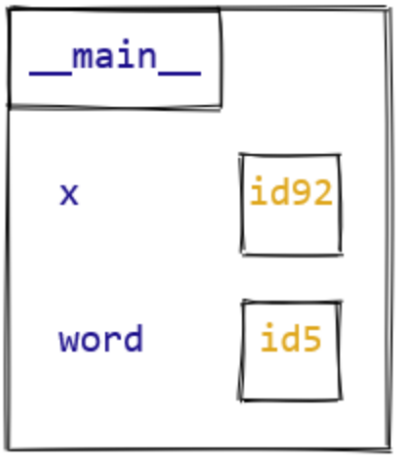
```
>>> x = 3
>>> word = 'bonjour'
```

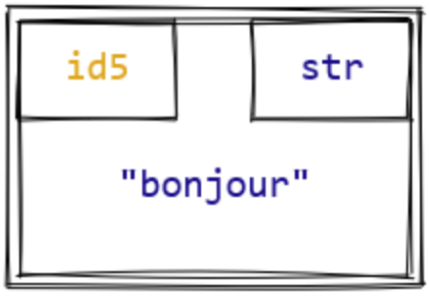
In our value-based memory model we would have represented these variables in a table:

`__main__`

Variable	*Value*
`x`	`3`
`word`	`'bonjour'`

With the full object-based Python memory model, we instead draw one table-like structure on the left showing the mapping between variables and object ids, and then the objects on the right. Each object is represented as a box, with its id in the upper-left corner, type in the upper-right corner, and value in the middle. The actual object id reported by the `id` function has many digits, and its true value isn't important; we just need to know that each object has a unique identifier. So for our drawings we make up short identifiers such as `id92`.





So there is no `3` inside the box for variable `x`. Instead, there is the *id* of an object whose value is 3. The same holds for variable `word`; it references an object whose value is `'bonjour'`.

Notice that we didn't draw any arrows. Programmers often draw an arrow when they want to show that one thing references another. This is great once you are very confident with a language and how references work. But in the early stages, you are much more likely to make correct predictions if you write down references (you can just make up id values) rather than arrows.

Assignment statements and evaluating expressions

You've written code much more complex that what's above, but now that we have the full Python memory model, we can understand a few more details for fundamental Python operations. These details are foundational for writing and debugging the more complex code you will work on this year. So let's pause for a moment and be explicit about two things.

Evaluating an expression. First, we said earlier that evaluating any Python expression produces a value. We now know that it is more precise to say that evaluating any Python expression produces *an id of an object representing the value of the expression*. Exactly what this object is depends on the kind of expression evaluated:

- If the expression is a literal, such as `176.4` or `'hello'`, Python creates an object of the appropriate type to hold the value.
- If the expression is a variable, Python looks up the variable. If the variable doesn't exist, a `NameError` is raised. If it does exist, the expression produces the id stored in that variable.
- If the expression is a binary operation, such as `+` or `%`, first Python evaluates the expression's two operands and applies the operator to the resulting values, creating a new object of the appropriate type to hold the resulting value. The expression produces the id of the new object.
- There are additional rules for other types of expression, but these will do for now.

Assignment statements. Second, we said earlier that an assignment statement is executed by first evaluating the right-hand side expression, and then storing it in the left-hand side variable. Here is a more precise version of what happens:

1. Evaluate the expression on the right-hand side, yielding the id of an object.
2. If the variable on the left-hand side doesn't already exist, create it.
3. Store the id from the expression on the right-hand side in the variable on the left-hand side.

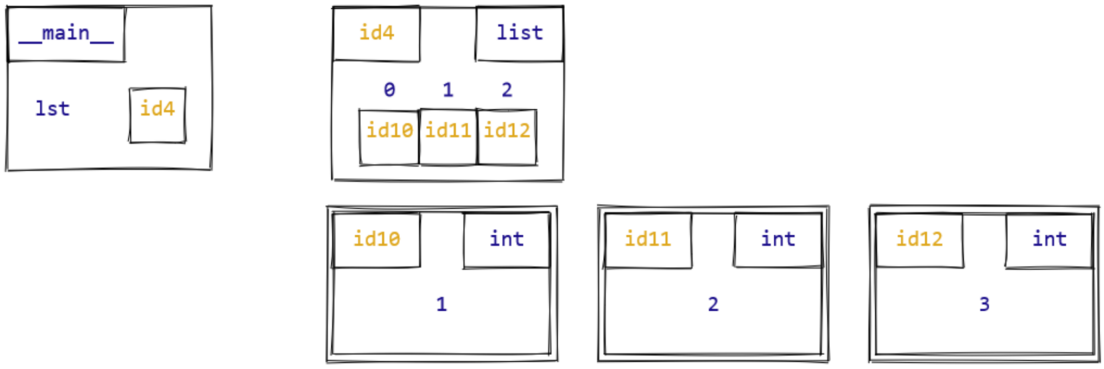
Representing compound data

So far, the only objects we've looked at in the Python memory model are instances of primitive data types. What about compound data types like collections and data classes?

Now that we have our object-based memory model, we are in a position to truly understand how Python represents these data types. *An instance of a compound data type does not store values directly; instead, it stores the ids of other objects.*

Let's see what this means for some familiar collection data types.

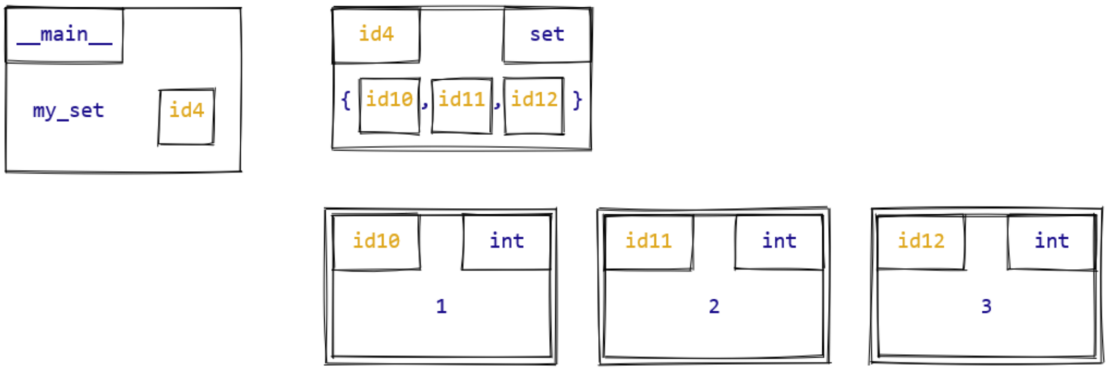
- *Lists*. Here is an object-based memory model diagram showing the state of memory after executing `lst = [1, 2, 3]`.



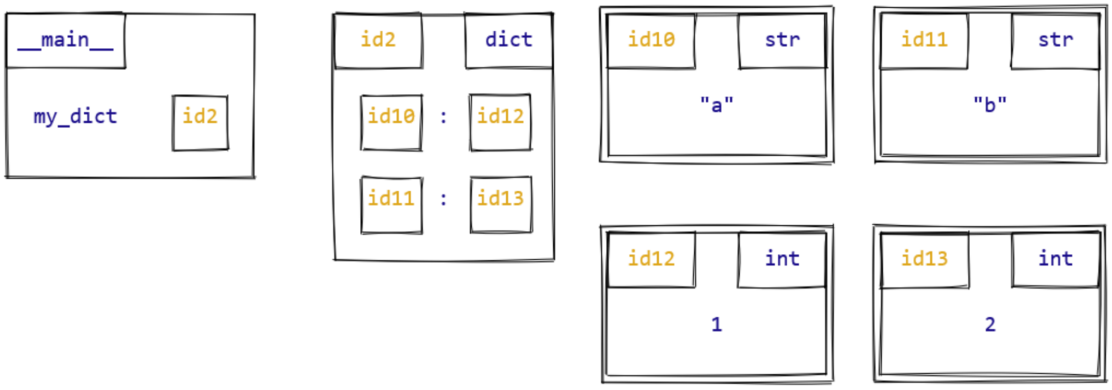
  Notice that there are four separate objects in this diagram: one for the each of the `ints` `1`, `2`, and `3`, and then one for the `list` itself.[2]

  [2] This illustrates one of the trade-offs with the Python memory model. It is more accurate than our value-based memory model, but that accuracy comes at the cost of having more parts and therefore more time-consuming to create.

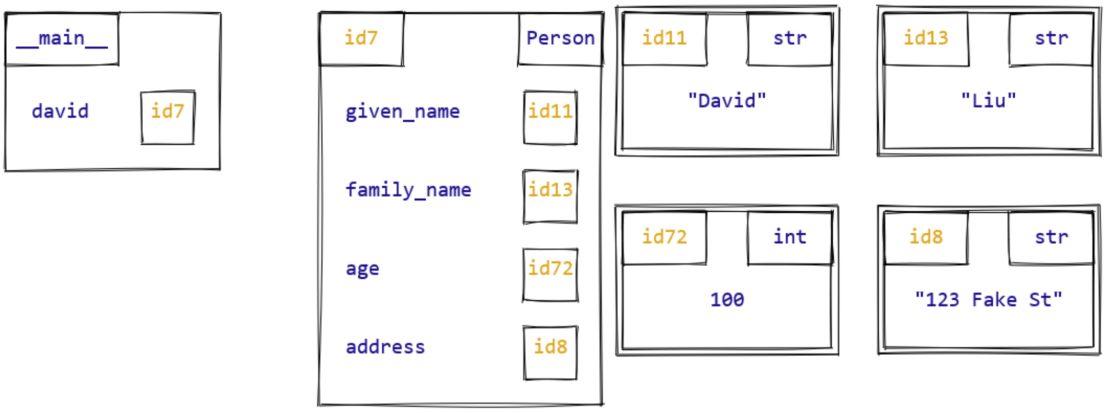
- *Sets*. Here is an object-based memory model diagram showing how Python represents the set `my_set = {1, 2, 3}`.



- *Dictionaries*. Here is an object-based memory model diagram showing the dictionary `my_dict = {'a': 1, 'b': 2}`. There are five objects in total!



- *Data classes*. All Python data classes are compound data types, and instances also store the ids of other objects. Unlike the collection data types we looked at above, these ids are not bundled in a collection, but instead each associated with a particular instance attribute. Here is how we represent our favourite `Person` object.



You may have noticed one difference between how we drew the object boxes of the primitive vs. compound data types above. We will use the convention of drawing a *double box* around objects that are immutable. Think of it as signifying that you can't get in there and change anything.

Visualizing variable reassignment and object mutation

Our last topic in this section will be to use our object-based memory model to visualize variable reassignment and object mutation in Python.

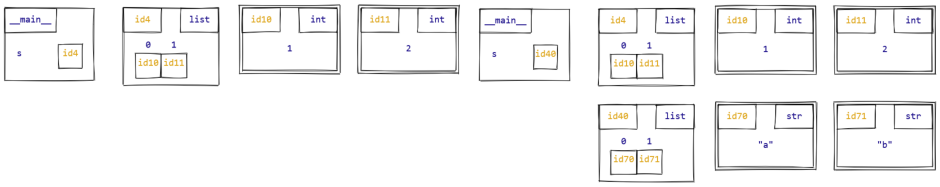
Consider this simple case of variable reassignment:

```
>>> s = [1, 2]
>>> s = ['a', 'b']
```

Here is what our memory model looks like after the first and second lines execute:

Before reassignment	**After reassignment**

Before reassignment **After reassignment**



Using this diagram, we can see what happens when we execute the reassignment `s = ['a', 'b']`: a new `list` object `['a', 'b']` is created, and variable `s` is assigned the id of the new object. The original list object `[1, 2]` is not mutated. Variable reassignment *does not mutate any objects*; instead, it changes what a variable refers to. We can see this in the interpreter by using the `id` function to tell what object `s` refers to before and after the reassignment:

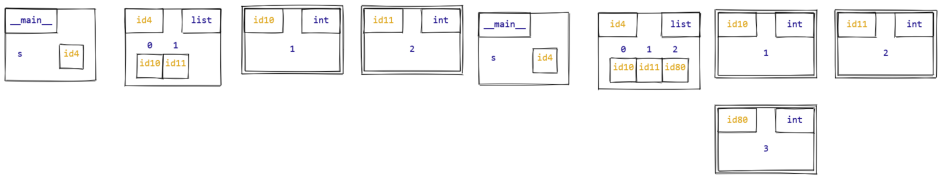
```
>>> s = [1, 2]
>>> id(s)
1695325453760
>>> s = ['a', 'b']
>>> id(s)
1695325453248
```

Notice that the ids are different, indicating that `s` refers to a new object.

Contrast this with using a mutating `list` method like `list.append`:

```
>>> s = [1, 2]
>>> list.append(s, 3)
```

Before mutation **After mutation**



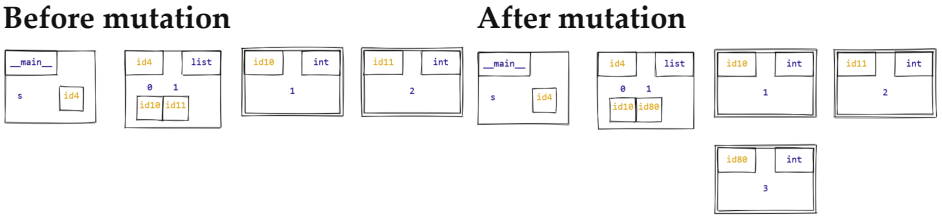
In this case, no new `list` object is created, though a new `int` object is. Instead, the list object `[1, 2]` is mutated, and a third id is added at its end. Note that even changing the list's size doesn't change its id! Again, we can verify that `x` refers to the same `list` object by inspecting ids:

```
>>> s = [1, 2]
>>> id(s)
1695325453760
>>> list.append(s, 3)
>>> id(s)
1695325453760
```

And finally, one last example that blends assignment and mutation: assigning to part of a compound data type. Consider this code:

```
>>> s = [1, 2]
>>> s[1] = 300
```

What happens in this case?



The statement `s[1] = 300` is also a form of reassignment, but rather than reassigning a variable, it reassigns an id that is part of an object. This means that this statement *does* mutate an object, and doesn't reassign any variables. We can verify that the id of `s` doesn't change after the index assignment.

```
>>> s = [1, 2]
>>> id(s)
1695325453760
>>> s[1] = 300
>>> id(s)
1695325453760
```